A Programmable Router Architecture Supporting Control Plane Extensibility

Jun Gao†, Peter Steenkiste†, Eduardo Takahashi‡, and Allan Fisher†
† School of Computer Science, ‡Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA
Email: {jungao, prs, takahasi, alf}@cs.cmu.edu

Abstract

The Internet is evolving from an infrastructure that provides basic communication services into a more sophisticated infrastructure that supports a wide range of electronic services such as virtual reality games and rich multimedia retrieval services. However, this evolution is happening only slowly, in part because the communication infrastructure is too rigid. In this paper, we present a programmable router architecture, in which control plane functionality of the router can be extended dynamically through the use of *delegates*. Delegates can control the behavior of the router through a well defined router control interface, allowing service providers and third-party software vendors to implement customized traffic control policies or protocols. We describe Darwin, a system that implements such an architecture. We emphasize the runtime environment the system provides for delegate execution and the programming interface the system exports to support delegates for traffic control and resource management. We demonstrate the advantages of using this system by presenting several delegate examples that can improve the quality of execution of end-user applications or create new network control policies.

1 Introduction

The Internet has evolved from a basic bitway pipe to a more sophisticated infrastructure that supports electronic services. The services today are fairly primitive and are typically related to collecting information over the web. Richer services such as high quality video conferencing, virtual reality games, and distributed simulation have been promised. Progress is slow in part because the infrastructure is inflexible. Routers are closed boxes that execute a restricted set of vendor software. Compute and storage servers are typically dedicated to support one type of services. An alternate architecture is to have an open infrastructure in which specific services can be instantiated on demand, much like what we do on a PC. One of the advantages of this approach is that it allows a larger community of people to develop services, which spurs innovation. Let us use some examples to motivate this approach.

A first class of examples addresses the customization of traffic control and management. Today, the range of traffic management options is fairly limited. While switches and routers increasingly have some support for classification and scheduling, these capabilities are often only used in simple ways, e.g., to filter out certain types of traffic, to do some simple prioritization of flows, or to implement standardized QoS mechanisms such as DiffServ. One could envision that these mechanisms could be used by users to handle their traffic in specific ways. For example, one service provider could implement gold/silver/bronze service differentiation in a proprietary way, while another service provider implements communication services with stronger guarantees. Similarly, one could envision deploying a Virtual Private Network(VPN) service, in which VPNs can use different traffic control policies or can even deploy different control protocols.

The second class of examples consists of value-added services, i.e., services that require not only communication, but also data processing and access to storage. Examples include video conferencing with video transcoding and mixing support, customized web searching services, and application-specific multicast. While it is possible to deliver these services using a set of dedicated servers, efficiency could be improved if the service can be deployed dynamically on servers that are leased on an "as needed" basis. This would allow the service to adapt to the demands and to the locations of the customers. Value-added services can also benefit from customized traffic management support. For example, a virtual reality game service provider may want to handle control, audio, and video traffic flows in different ways. This may require customized traffic control policies on the router.

As long as routers are closed boxes that are shipped with a set of standard protocols, it is unlikely that these examples will be realized. The above examples can best be supported by a programmable network infrastructure. The network offers computing, storage, and communication resources that can be allocated and programmed to deliver a specific service. Many standards exist to access remote storage servers (web, file systems, databases) and the operating systems that run on compute servers have well-defined interfaces that have been specifically defined to support a wide range of applications. Some aspects of the operating system interface have also been standardized (e.g., POSIX). Routers, which we can view as communication servers, are not programmable today, and router vendors typically offer a small fixed set of control software. In this paper we present an alternate router architecture, in which the control plane functionality of the router can be extended using "delegates", code segments that implement customized traffic control policies or protocols. Delegates can affect how the router treats the packets belonging to a specific user through the Router Control Interface, or RCI. With this architecture, a broader community, e.g., third party software vendors or large providers of value-added services, can develop applications for routers.

The remainder of the paper is organized as follows. We first define a network architecture in Section 2 in which network functions (e.g., QoS) can be selectively customized to meet the special needs of users. In Section 3, we describe Darwin, a specific instance of the above architecture. We focus on the runtime environment for delegates, code segments that can customize network control and management. In Section 4 we present some examples of how delegates can be applied to address a variety of resource management problems, and we discuss security issues raised by the use of delegates in Section 5. We present related work in Section 6 and summarize in Section 7.

2 Programmable Network Architecture

We first characterize the network programmability requirements and introduce the concept of a "delegate". We then present a programmable network architecture that can support delegates.

2.1 Network programmability

We can distinguish between two types of operations on data flows inside the network. The first class involves manipulation of the data in the packets, such as video transcoding, compression, or encryption. Since most routers do not have significant general-purpose processing power, this type of processing will typically take place on compute servers, e.g., workstation clusters inside the network infrastructure. In the future, these tasks could also be performed on routers that have been specifically built to also perform data processing, besides the usual routing functions. The second class of operations on data flows changes how the data is forwarded but typically does not require processing or even looking at the body of packets. Examples include changing the bandwidth allocation of a flow, tunneling, rerouting, and selective packet dropping. The nature of these operations is such that they are best executed on routers or switches.

We will call the code modules that perform these tasks "delegates" since they represent the owner of the data flows inside the network. Data delegates perform data processing operations and execute on compute servers or specially designed routers. Control delegates execute on routers and are involved in the control of data flows. This simple classification of delegates is somewhat artificial in the sense that some delegates may fall in between these two classes. For example, a delegate that is concerned with control may need to look at the packet body to decide how to handle packets. Similarly, a "mostly control" delegate may on rare occasions have to modify a packet. Nevertheless, the distinction is useful because the two classes of delegates impose very different requirements on the system they run on. Control delegates require an environment that provides a fairly rich set of mechanisms to control data flows, while data delegates must run on a platform that has substantial computational power.

While nobody is likely to argue against the use of data delegates on compute servers for data processing, the need for control delegates is less obvious. One could imagine routers with fixed functionality, similar to today's routers, where users can control how their traffic flows are handled by passing parameters to the routers using a signalling protocol. The examples discussed in Section 1 provide some motivation why directly executing code (i.e., control delegates) on the router may be a more effective way to customize traffic control and management. A first reason is that control delegates can respond much more quickly to changes in the traffic conditions; it would take an entity at the edge of the network at least one and more likely several round-trip times before it can first observe and then respond to a change in the network. A second reason is that it seems impractical to identify all possible user requirements *a priori*, so that they can be addressed by the default router software; an architecture based on delegates is more flexible and extensible. Finally, supporting customization by extending router functionality may often be a more natural and thus less error-prone solution. For example, if a service provider wants to use a routing protocol that is optimized for its traffic, doing so from the edges of the network is likely to be unnecessarily complicated.

In the rest of this paper we will focus on the design of a router architecture that can host control delegates. We simply use the term "delegate" if no confusion is possible.

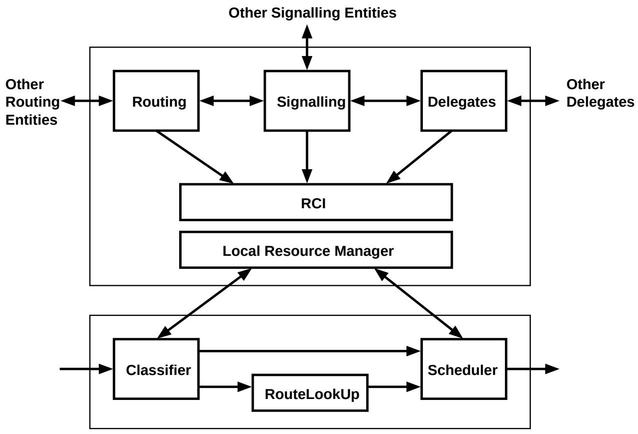


Figure 1: Node architecture

2.2 Router architecture

Figure 1 presents a node architecture through which delegates can be added to a router. The architecture shows a control plane that executes control protocols (e.g., routing, signalling) and a data plane that is responsible for packet forwarding. (Control) delegates execute in a special runtime environment that is part of the control plane. This design is motivated both by the intended use of control delegates (control and management of traffic flows) and by practical design considerations (we do not want to add unnecessary complexity to the data forwarding path, where speed and simplicity are critical). On some routers it may also be possible to insert data delegates in the data forwarding plane.

Control delegates can change how traffic is handled in the data forwarding plane through the Router Control Interface (RCI). The RCI provides a set of operations on *flows*, a sequence of packets with a semantic relationship defined by applications and service providers. Flows are defined on each router using a flow spec [5], i.e. a list of constraints that fields in the packet header must match for that packet to belong to the flow. The classifier uses the flow specs to determine what flow incoming packets belong to. Classification should take place early in the data forwarding path so that packets can be handled appropriately. Some flow-specific actions, such as tunneling and rerouting, are best executed on the input interface. Actions on the output interface are generally associated with QoS related parameters so that after classification the scheduler can schedule the packets belong to each class accordingly to meet their QoS requirements.

We can view the RCI as an instruction set that operates on flows as a basic data type. A critical design decision in the architecture is the definition of this interface, i.e., what functions should be exported to the delegates. The RCI should be broad enough to support both value-added services and network management applications. However since the RCI will be used by control delegates on routers, efficiency and security issues should also be taken into account when defining the RCI. By examining a wide range of services, we identified the following basic functions for RCI: methods for delegates to manipulate flows, to collect network information, to perform resource management, and affect flow forwarding. Support for delegates to communicate with peer delegates are also included in RCI. In next section we will discuss in detail a particular implementation of RCI in the Darwin system.

Most of the components in the proposed router architecture can be found in many framework architectures designed to support QoS in the Internet. For example, the packet classification and scheduling modules are present in the IETF IntServ model [4, 12] and the more recent Differentiated Services model [20], [2]. The difference lies in that this architecture provides programmability in the control plane through the use of delegates and the programming interface, RCI. A more detailed comparison with related work can be found later in the paper.

3 Darwin Delegates

We give a brief overview of the Darwin system and describe the Darwin RCI and delegate runtime environment.

3.1 Darwin delegate design

The Darwin project developed a set of *customizable* resource management mechanisms. Customizability allows applications and service providers to tailor resource management, and thus service quality, to fit their needs. Darwin includes three mechanisms that operate on different time scales. A resource broker, called Xena, selects resources that meet application needs using application-specified metrics to optimize resource use [8]. Delegates support customizable runtime resource management, as described above. Finally, Darwin uses a hierarchical packet scheduler that supports a wide range of policies and integrates per-flow QoS and link sharing in a single framework [21]. The activities of the three mechanisms are coordinated by a signalling protocol called Beagle [9].

Darwin delegates are based on the architecture outlined in the previous section, but the architecture is extended in two ways. First, in Darwin the classifier that identifies flows does not only use the standard fields in the IP and transport header (IP address information, port numbers, and protocol identifier), but also an optional application identifier. This allows services to define flows based on their own semantics. An example would be layered video, where different layers are tagged with a different application identifier. In our current implementation, the application identifier is stored in the packet as an IP option. Other formats, e.g., the IPv6 flow ID, are possible. Every packet should normally be classified once, soon after it arrives on the router. In Darwin, for implementation convenience, we use two classification engines, one at the input port and the other at the output port. The extra classification adds negligible overhead in our system.

Second, as mentioned earlier, Darwin uses hierarchical resource management. This means that the resource distribution of a link is rep-

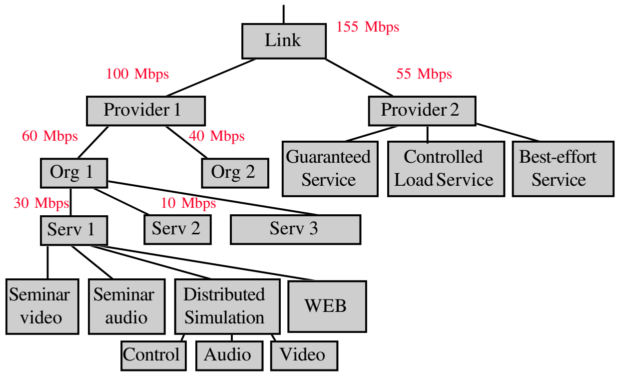


Figure 2: Example resource tree

resented by a resource tree (Figure 2), with the root representing the link, leaf nodes actual data flows, and interior nodes organizations, services or applications that control the flow or flow aggregate corresponding to their children. Resource allocation policies can be specified for both leaf and interior nodes, so both per-flow QoS and link sharing can be supported in the same framework. This can be viewed as a "divide-and-conquer" approach to resource management. The bandwidth of a link (root node) can be divided across a set of organizations (children of the root), each of which can manage its bandwidth share by constructing an appropriate subtree. Darwin uses the Hierarchical Fair Service Curve scheduler [21], which has excellent isolation properties, i.e. changes in the structure or policies of one subtree do not affect the way traffic controlled by other subtrees is handled.

The combination of delegates (in the control plane) and hierarchical resource management (in the data plane) provides an excellent framework for the customization of traffic control and management. We emphasized earlier that one use of delegates is to customize how a specific set of flows is handled. This is achieved by associating a delegate with a specific node in the flow hierarchy(Figure 3), i.e. the delegate can only operate on flows associated with node and its subtree and cannot deal with other flows. In other words, the hierarchical scheduler provides the isolation of network resources in the data plane and the matching hierarchical delegates provide the isolation of traffic management and control in the control plane. Note that some delegates may operate on the network as a whole; such delegates are logically associated with the root.

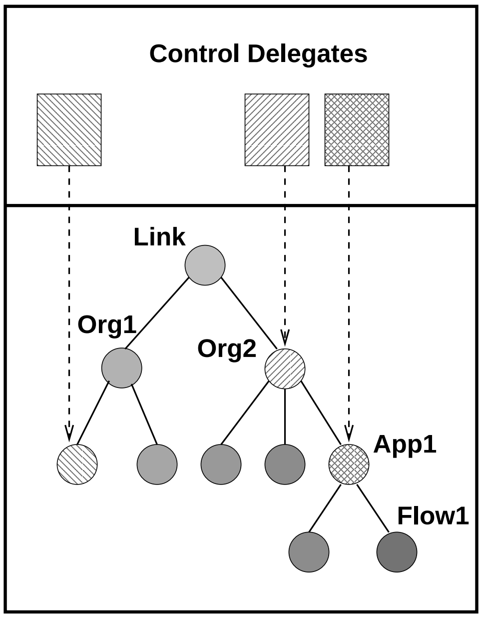


Figure 3: Pairing customization in the control and data plane

3.2 Router Control Interface

We describe five categories of functions that are necessary for the RCI to support a broad spectrum of delegates.

3.2.1 Flow manipulation methods

The RCI presents delegates with a flow-based programming model. The first class of functions allows delegates to define and manage flows. These calls will typically update the classifier data structures so it classifies packets accordingly. As mentioned above, in Darwin flows are defined based on a flow specification that includes the traditional IP and transport header fields and an application identifier. Once a flow has been defined, operations on these flows, e.g., specifying QoS parameters, or flow redirection, can then be performed. Four methods are available.

add_flow (*interface_name*, *flow_spec*) installs a flow to the given interface's flow hierarchy and returns a *flow_id* to the caller;

delete_flow (*interface_name*, *flow_id*) removes the flow with *flow_id* from the specified interface's flow hierarchy;

modify_flow (*interface_name*, *flow_id*, *new_flow_spec*) changes the flow spec of the flow with *flow_id* to *new_flow_spec* and returns a new *flow_id*;

retrieve_flow_hierarchy (*interface_name*, *flow_id*) returns the flow hierarchy tree rooted at the specified flow on the specified interface.

3.2.2 Resource management methods

Delegates can change how resources are allocated and distributed across flows by modifying states in the scheduler through the RCI. The scheduler relies on its hierarchical resource tree to schedule packets to meet each flow's QoS requirement. A node in the resource tree is identified by a node_id, and may correspond to multiple flows in the flow hierarchy. Each resource tree node is assigned a set of QoS parameters to specify the service that the flows corresponding to this node will receive. For the HFSC scheduler used in Darwin, the following functions are needed:

create_node(*interface_name*, *parent_node_id*) creates a child node for the node with *parent_node_id* and returns a *node_id*;

delete_node(*interface_name*,*node_id*) deletes the node with *node_id* from the resource tree;

add_service(*interface_name*,*node_id*, *service_parameters*) adds *service_parameters* to the node with *node_id*, i.e., flows that belong to this node will receive service specified by *service_parameters*;

change_service(*interface_name*,*node_id*, *new_service_parameters*) changes service parameters of the node with *node_id* to *new_service_parameters*;

add_flow_to_node(*interface_name*,*node_id*, *flow_id*) adds the flow with *flow_id* to the set of flows that belong to the node with *node_id*. This basically grants the flow with the service that the node has;

del_flow_from_node(*interface_name*,*node_id*, *flow_id*) removes the flow with *flow_id* from the set of flows that belong to the node with *node_id*. By calling this method, a flow's service can be revoked;

retrieve_tree(*interface_name*, *node_id*) returns the resource tree hierarchy rooted at the node with *node_id*.

Note that the precise nature of this class of functions depends on the scheduler, although the functions listed here should representative of most hierarchical schedulers, e.g. CBQ.

3.2.3 Flow redirecting methods

Delegates may want to reroute a flow's packets using a different route from the route that is given by using the default routing table, e.g., to avoid "hot" spots in the network. In some other applications delegates may need to tunnel packets to a different destination, e.g., for server load balancing or special data processing. For these purposes, RCI offers the following methods to delegates for flow redirecting.

tunnel(*interface_name*, *flow_id*, *encap_src_addr*, *encap_dst_addr*) prepends an extra header with *encap_src_addr*, *encap_dst_addr* as source and destination address to the packets belong to the flow with *flow_id*. This function for example can be used to redirect flows to a data delegate which resides in a compute server;

reroute(*interface_name*, *flow_id*,*next_hop_addr*) enables data plane to skip routing table look up and forward packets out using the interface that corresponds to *next_hop_addr*;

drop(*interface_name*, *flow_id*) drops a flow's packets at the input port;

get_route(*dst_addr*) retrieves next hop address from the routing table for *dst_addr*;

change_route(*dst_addr*, *next_hop_addr*) updates the next hop address in the routing table entry for *dst_addr* to *next_hop_addr* through a routing socket, and this requires superuser privilege.

In our implementation, the flow redirecting methods are realized as actions and they are associated with flows. After a packet is classified to a flow, the action if any that is associated with this flow will be taken on this packet. Other ways of doing flow redirecting, such as building separate routing tables for flows, are also considered. We are also working on mechanisms to support adding application specific actions dynamically as plug-ins.

3.2.4 Traffic monitoring and queue management methods

Delegates can monitor network congestion status by examining queue lengths. Bandwidth usage and delay data of each flow are important for delegates to make adequate reacting decisions. The hierarchical scheduler implements a fairly sophisticated queuing discipline, and RCI exports basic queue management methods to delegates. RCI also allows delegates to selectively drop a flow's packets.

monitor_on(*interface_name*) turns on the monitor mode in the kernel to record bandwidth usage and delay data;

monitor_off(*interface_name*) turns off the monitor mode;

retrieve_data(*interface_name*, *flow_id*) returns the bandwidth usage and delay data of the flow with *flow_id*;

probe(*interface_name*, *flow_id*) returns the queue size of the queue corresponding to the flow with *flow_id*;

set(*interface_name*, *flow_id*, *queue_parameters*) sets the queue parameters, such as high and low thresholds for the flow with *flow_id*;

request(*interface_name*, *flow_id*, *timeout*) when called by the delegate, delegate thread will be blocked and restarted when either timer times out or receives congestion notification from the kernel. The kernel experiences congestion when the packets belong to the flow with *flow_id* are dropped;

3.2.5 Support for delegate communication

Delegates can set up communication channels among them to coordinate activities with peers on other routers and to interact with the application on endpoints. Messaging between delegates allows the system to both gather global knowledge and perform global actions, as in the case of rerouting for load balancing and most of the network management applications. Interaction with applications on end-points increases the flexibility of the system, as adaptation to network events typically involves the sources. The inter-delegate communication, is often application specific, i.e., different applications require delegates to exchange different information. We built the communication channels between delegates in our examples on top of the standard `java.net` classes.

3.2.6 Other methods

The above five classes of functions are likely to be appropriate for most routers. However, individual routers may have additional functionality on their data forwarding path and they may allow delegates to control these functions by calls to the router control interface. As an example, on a router that supports RED, delegates may be able to change the thresholds used to trigger early packet drops. On a DiffServ edge router, delegates may need access to the control parameters for shapers and meters. Other routers may have algorithms to identify non-conformant flows and may allow delegates to get access to this information; we will give an example using this functionality in Section 4. Clearly, an interface standard would have to be extensible, so new calls can be added as technology evolves, not unlike the way MIB definitions for network management.

3.3 Delegate implementation

Darwin delegates are based on Java and use the JDK1.2 Java virtual machine from Sun Microsystems [15], capable of just-in-time (JIT) compilation and available for many platforms. This environment gives us acceptable performance, portability, and safety features inherited from the language. Delegates are executed as Java threads inside the virtual machine "sandbox". A delegate is characterized by its QoS requirement (e.g., the amount of CPU and memory resources needed), and runtime environment needed (e.g., Java, Perl, VisualBasic script, etc.).

Experiments to measure the overhead of the RCI calls from within the delegate runtime environment showed minimal difference between calls from Java delegates and calls from equivalent C programs. That is a reasonable result since RCI calls are actually implemented as native methods. The overhead of most delegates calls in an unloaded system is measured to be around 5 microseconds using machines in our testbed (see Section 4). As the system load increases (e.g., with packet forwarding) the system call latency becomes highly variable and unpredictable since our operating systems do not offer real-time guarantees.

Delegates are installed through the Darwin signalling protocol, Beagle, using a multi-step process. First, the application or service provider submits delegate Java bytecode, delegate resource and runtime requirements, together with a list of flow specs to Beagle. Second, Beagle transports this information to Beagle daemons running on the relevant routers. Third, each Beagle daemon performs local admission control for both the flows and (if necessary) any delegates. For delegates, this involves verifying that the delegate runtime environment has the required libraries to support the delegate. At this point, Beagle would also verify that the router has sufficient CPU and memory resources to accommodate the delegate, but since our environment (PC-based routers) can not explicitly manage these resources, this step is not implemented. Fourth, if admission control succeeds, Beagle will first set up the flows by making appropriate calls to the classifier and scheduler. Finally, Beagle will pass the byte code to the delegate runtime environment, start up the delegate, and pass the delegate a set of handles identifying the flows it is responsible for. The interface that is used by Beagle to start delegates is described in more detail elsewhere [9].

Delegates written in C are also supported in our system but in a much more restrictive way. Data delegates that run on compute servers can be set up in a similar fashion.

Delegates are a very focused application of active networking [23]: they are installed asynchronously from the rest of the data traffic by a separate signalling protocol on a per-application or per-service basis; a delegate can only operate on the flows that it is associated with. As is the case with active networking in general, delegates raise significant safety and security concerns. In Section 5 we will discuss safety and security issues related to delegates in more details.

4 Examples

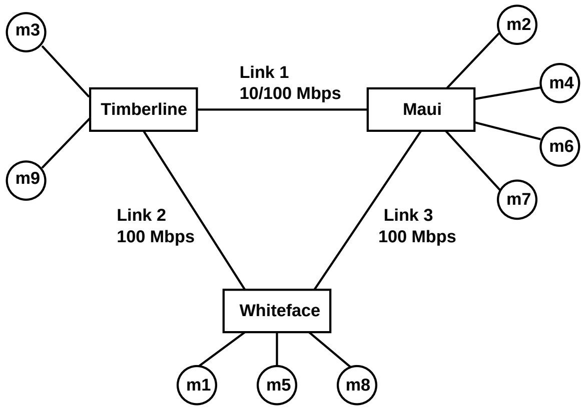


Figure 4: Darwin IP testbed topology

The Darwin system has been implemented on FreeBSD and NetBSD PC routers. Experiements were performed to test the system on a local testbed shown in Figure 4. The three routers, shown in boxes, are Pentium II 266 MHz PCs running Darwin kernel which is built on top of FreeBSD 2.2.6. The end systems `m1` through `m9` are Digital Alpha 21064A 300 MHz workstations running Digital Unix 4.0. All links are full-duplex point-to-point Ethernet links configurable as either 100 Mbps or 10 Mbps. Unless specified, the links are configured as 100 Mbps. In this section, we present five delegate examples and show some experimental results obtained from this testbed. The first three examples demonstrate how application specific services can be added to the network through delegates to improve the quality of execution of these applications. The last two examples show that delegates can dynamically customize traffic control and management in a network.

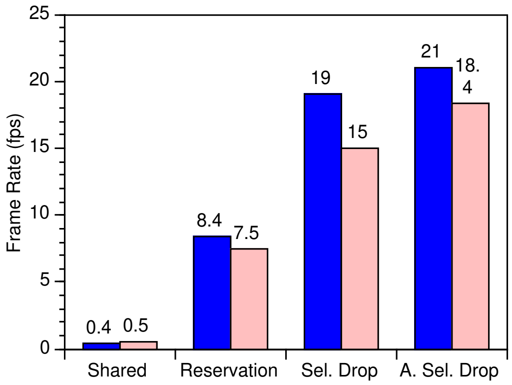


Figure 5: Video quality under four scenarios

4.1 Selective packet dropping for MPEG video streams

MPEG video streams are very sensitive to random packet loss because of the dependencies between three different frame types: I frames (intracoded) are self contained, P frames (predictive) use a previous I or P frame for motion compensation and thus depend on this previous frame, and B frames (bidirectional-predictive) use (and thus depend on) previous and subsequent I or P frames. Because of these inter-frame dependencies, losing I frames during transmission is extremely damaging, while B frames are the least critical. Under congestion, the received video quality is degraded due to packet loss but the quality can be enhanced if only B frames are dropped at the congested link instead of random frames (including crucial I frames) being dropped.

Delegates can be used to perform selective packet dropping on congested routers. Delegates are injected into routers that may potentially be congested, and these delegates are associated with MPEG flows in the data plane. In the resource hierarchy, a flow is set up to correspond to the MPEG application and three sub-flows differing from one another in the application identifier field in the flow spec are then added to differentiate the three different frame types. Packets belonging to different frame types are marked with specific application identifiers so that they will be classified into different sub-flows. Delegates monitor the queue length of the MPEG flow, and when congestion is detected (e.g., the queue length exceeds some preset threshold), delegates will instruct the data plane to drop the packets belonging to the B frame sub-flow, thus protecting I and P frames from being dropped due to queue overflow. As a result, the receiver will observe a gracefully degraded quality.

Several experiments were performed on the testbed to demonstrate the effectiveness of selective packet discarding delegates. Three streams are sent over the Timberline-Maui link (bottleneck link) of the testbed: two MPEG video streams and one unconstrained UDP stream. The UDP stream causes congestion on this link. Both video sources send at a rate of 30 frames/second, and our performance metric is the rate of correctly received frames. Figure 5 compares the performance of four scenarios. In the first scenario, the video and UDP data packets are treated equally, and the random packet losses result in a very low frame rate, as expected. In the second case, the video streams share a bandwidth reservation which equals to the sum of the average video bandwidths. This improves performance, but the video streams are bursty, and random packet loss during peak transfers results in less than a third of the frames being received correctly. In the third scenario, we place a delegate on Timberline. As described earlier, the delegate monitors the length of queue used by video streams and drops the B frames when congestion is observed. Packet dropping is switched off when the queue size drops below a lower threshold. Figure 5 shows that is quite effective: the frame rate roughly doubles.

While delegates provide an elegant way of selectively dropping B frames, the same effect could be achieved by associating different priorities with different frame types. In scenario four we use a delegate to implement a more sophisticated customized drop policy. In scenario three, either all or none of the B frames are dropped. By dropping the B frames of only a subset of the video streams, we can achieve finer grain congestion control. The advantage of having a delegate control selective packet dropping is that choice of applications that should be degraded first can be customized. Scenario four in Figure 5 shows the results for a simple "time sharing" policy, where every few seconds the delegate switches the stream that has B frames dropped. This improves performance by another 10-20%. Policies that differentiate between flows could similarly be implemented.

4.2 Dynamic control of MJPEG video encoding

An alternative to selective frame dropping to deal with congestion is to use a video transcoder to compress, or change the level of compression, of the video stream depending on the available bandwidth. It is possible to optimize video quality by dynamically adjusting the transcoder's level of compression. We use both control and data delegates in this example to illustrate how delegates can control compression levels for video quality optimization via flow monitoring, flow redirecting and inter-delegate communication. A control delegate is set up on the router before the bottleneck link on the route of the video application. The control delegate alters the original route that the video stream will take by first redirecting the flow to a data delegate which resides on a compute server next to the router. The data delegate functions as a transcoder in that it takes in raw video and generates MJPEG frames using different compression level. The MJPEG frames are then fed back to the bottleneck router and will only then be forwarded out to the original intended receiver. The control delegate monitors the bottleneck link's congestion status and available bandwidth in a similar way as the previous example. When facing congestion, the control delegate will inform the data delegate to use a higher compression level for less bandwidth usage. In other times, when the control delegate sees abundant bandwidth on the bottleneck link, it can prompt the data delegate to deploy a lower compression level for better video quality. This allows the video flow to opportunistically take advantage of available bandwidth whenever possible.

In the experiment, an application consisting of two MJPEG video streams and two bursty data streams is competing for network bandwidth with other users, modeled as an unconstrained UDP stream. All flows are directed over the 10 Mbps Timberline-Maui link. The application has 70% of the bandwidth, 20% for video and 50% for data, and the remaining 30% bandwidth of the link is for the competing users. The application's data streams belong to a distributed FFT computation. The data traffic is very bursty, since FFT alternates between compute phases, when there is no network usage, and communication phases, when the nodes exchange large data sets. An important property of the hierarchical link-sharing scheduler used in this experiment is that the video flows have priority on taking bandwidth not used by the FFT flows. This means that video quality can be improved significantly during the compute phases of the FFT, if the video can make use of the additional bandwidth.

A control delegate is placed on router Timberline and a data delegate is installed on server m9. So the video traffic when received by Timberline is forwarded to m9 first and after data delegate processing, the MJPEG frames are sent back to Timberline. Timberline then forwards these frames out on the Timberline-Maui link. Figure 6 shows a screen shot of the bandwidth used by the video flows (light grey) and FFT (dark grey). During FFT bursts, bandwidth is limited (20% of the link) and video quality is low, but between FFT bursts the video can use almost 70% of the link, resulting in increased video quality. Figure 7 shows a histogram of the received frame quality. We see that the majority of frames are received with either maximum quality of 100 (received when the FFT is in its computation phase) or with the minimum quality of 0 (when FFT is in its communication phase). Frames received with other quality settings reflect the ramp up and ramp down behavior performed by the control delegate as it tracks the available bandwidth.

4.3 Load-sensitive flow rerouting

Routing decisions in the Internet today are mostly load insensitive and application independent; in other words, the path taken by a packet

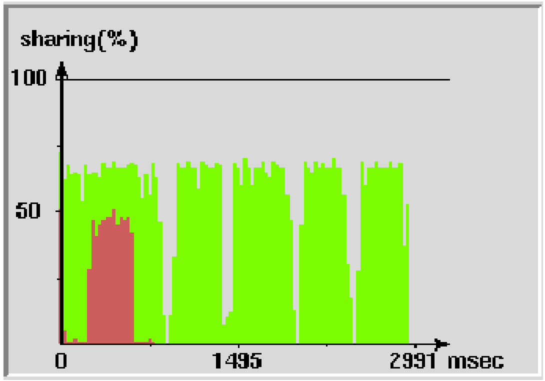


Figure 6: Bandwidth sharing between video and FFT streams.

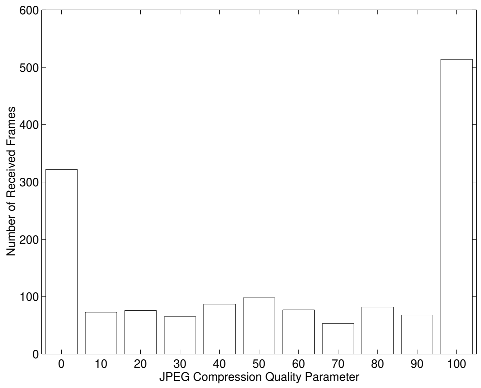


Figure 7: Distribution of received JPEG quality.

does not depend on the load in the network or the application the packet belongs to. While this results in simple and stable routing protocols, it can also cause inefficient use of network resources. Discovering a lightly-loaded route and using it to reroute an application's flow may significantly improve the application's performance.

Similarly in a client-server scenario, to handle multiple clients' requests, it is often necessary to have multiple servers. However, there are times that one server is overloaded by requests from clients for various reasons, and other servers are idle. In this case, it would make sense to have some mechanism inside the network to redirect some requests to the lightly-loaded servers to achieve better overall performance.

Delegates are equipped with the capability of rerouting a flow's packets or even redirecting a flow to a different destination through tunneling. Here we first explain how flow rerouting can be done using multiple delegates and then show a proof-of-concept experiment performed on the testbed. Flow redirecting can be implemented in a similar fashion. For an application that has multiple flows and involves multiple endpoints, necessary resources are reserved on the routers along the default path that each flow will take. Delegates corresponding to the application's flows are installed on these routers. The application's endpoints, related routers and links, along with the delegates form a "virtual mesh" structure. We observe that all the application's flows may not be active at the same time, and this offers opportunities for delegates to discover a less used route within the "virtual mesh" for current active flows. A delegate monitors the bandwidth usage of this application's flows on its router's links and periodically queries its peers for bandwidth usage on remote routers. Once a better route in terms of available bandwidth is found, each delegate along the new route will change the forwarding behavior locally to use the interface that belongs to the new route. Note that in this scheme, the route changes are within the application's "virtual mesh" and will not affect other traffic's default forwarding.

We use a simple experiment to illustrate how delegates can balance an application's traffic by rerouting. The example application has multiple flows that use the three links between the routers. The application reserves 50% of the bandwidth on each of these links. The application's flows originate from either m8 or m9 and the resource trees on Link 1, 2 and 3 are shown in Figure 8. On Link 1, Node 1 corresponds to this application's flows and Node 2 corresponds to some other competing application's flows. Node 3 corresponds to one specific flow of this application, m9 to m2. Node 4 corresponds to another flow of this application, m9 to m4, and is drawn in dotted lines, meaning this flow is not known to the scheduler and it will be classified to Node 1. On Link 2, Node 1 and Node 3 represent the same flows as on Link 1, but there are no other applications that have reserved resources. On Link 3, Node 1 again corresponds to our application, and Node 2 represents some other competing application. Nodes 3 and 4 correspond to a flow from m9 to m2 and a flow from m8 to m6 respectively, and they are drawn in dotted lines, meaning that they do not have individual reservations.

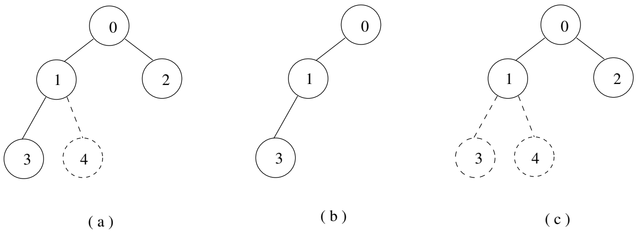


Figure 8: Resource trees used in the experiment in Section 5.5. (a) Resource tree on Link 1 (b) Resource tree on Link 2 (c) Resource tree on Link 3

The delegate on Timberline is responsible for one particular flow, m9 to m2, of this application. It knows the bandwidth usage by this flow on Links 1 and 2 by directly monitoring them, and it queries the delegate on Whiteface to get the available bandwidth for this application on Link 3. Initially, the route for flow m9 to m2 passes router Timberline and Maui only (the shortest path). Since the application has a 50% reservation, this flow gets about 50 Mbps throughput. When another flow, m9 to m4, which belongs to this application joins, they share the bandwidth reserved by the application, i.e, each gets about 25 Mbps. At this time, the delegate on Timberline knows that 50 Mbps are available on Link 2 and, by querying the delegate on Whiteface, it understands that the available bandwidth for this application on Link 3 is 50 Mbps. The minimum of these two numbers is larger than what flow m9 to m2 is using, so the delegate on Timberline makes the decision to reroute flow m9 to m2 through Whiteface. Later, when another flow, m8 to m6, which also belongs to the application starts, flow m9 to m2 still goes through Whiteface until flow m9 to m4 finishes, making more bandwidth available on Link 1. At that time, the delegate changes the route for flow m9 to m2 back to its initial route.

The results are shown in Figure 9. The series annotated with "Node 3, Link 1" and "Node 3, Link 2" show the throughput of flow m9 to m2 when it passes only Link 1, and when it passes Links 2 and 3 respectively. "Node 4, Link 1" represents the throughput of flow m9 to m4 and "Node 1, Link 3" represents the throughput of flow m8 to m6. We can see that flow m9 to m2 adapts to the link that has larger available bandwidth in a timely fashion.

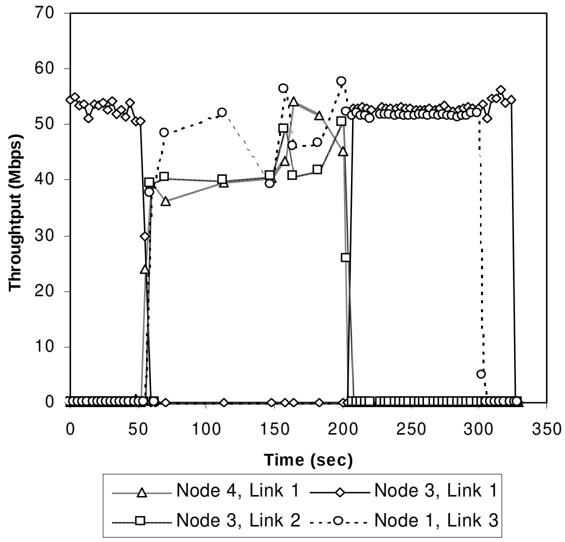


Figure 9: Load-sensitive rerouting results

4.4 Selective dropping of non-adaptive flows

Applications that do not use appropriate end-to-end congestion control are an increasing problem in the Internet. These applications do not back off when there is congestion, or they back off less aggressively than users that use correct TCP implementations, and as a result, they get an unfair share of the network bandwidth. We will refer to such flows as non-conformant. In response to this problem, researchers have developed a variety of mechanisms that try to protect conformant flows from non-conformant flows. These include Fair Queueing scheduling strategies that try to distribute bandwidth equally, and algorithms such as RED [13] and FRED [17] that, in case of congestion, try to selectively drop the packets of non-conformant flows.

While, once deployed, these mechanisms will improve the fairness of bandwidth distribution at the bottleneck link, they address only part of the problem since they are designed to work locally. The problem is that non-conformant flows still consume (and probably waste) bandwidth upstream from the congested link. Upstream routers may not respond to the non-conformant flow, for example because they have no support for detecting non-conformant flows, or because the flow cannot be detected (e.g. because of aggregation in a core router), or because the flow appears to be conformant (e.g. does not cause congestion). This problem can be addressed by having routers propagate information on the non-conformant flows upstream along the path of those flows. This approach can also deal with denial-of-service attacks on servers: the server can report the attack to the router it is attached to, and the network can then track down, and selectively drop, that flow potentially all the way to the source.

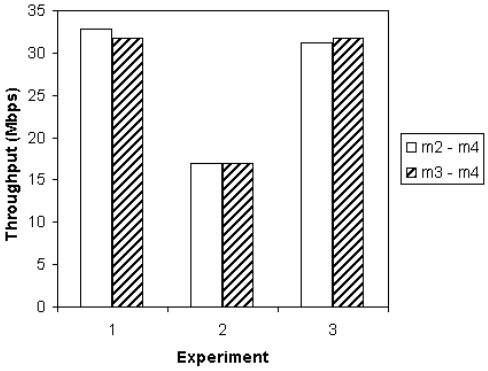
We implemented a simple version of this solution using delegates. A delegate locally monitors the congestion status and tries to identify non-conformant flows among the flows it is responsible for. In our implementation, a flow is considered to be non-conformant if its queue is overflowing for an extended period of time; more sophisticated mechanisms would be needed in a production version of the system. Once a "bad" flow has been identified, the delegate enables selective packet dropping for the flow, and sends the flow's descriptor to a peer delegate on the upstream router. When a delegate receives a report of a "bad" flow, it verifies that the flow indeed has a high bandwidth and enables selective packet dropping, if possible, and forwards the message to the upstream router. Clearly, many alternative policies could be implemented; for example, only a certain percentage of the packets could be dropped to reduce its bandwidth instead of dropping all packets as in our implementation.

We conducted two experiments to show the effectiveness of these delegates. In the first experiment, two TCP streams (m2-m4 and m3-m4) compete with a non-conformant UDP stream (m5-m4). The bottleneck link is Maui-m4. Throughputs of the TCP flows under different conditions are shown in Figure 10(a). In the first case, the UDP stream is switched off, and the two TCP flows share the link bandwidth evenly. In Experiment 2, the UDP flow is present in the background and the TCP flows' performance is greatly reduced because of the UDP flow. In Experiment 3, the delegates are running on all three routers. The UDP flow from m5 to m4 does not cause congestion on the link between Whiteface and Maui, and therefore the delegate on Whiteface does not react. However, there is congestion on the Maui-m4 link, and Timberline correctly identifies the UDP flow as non-conformant. It then informs the delegate on Whiteface through the delegate communication channel. Both routers then drop the UDP flow's packets, and as a result, the two TCP flows recover their throughputs to about the same level as in Experiment 1.

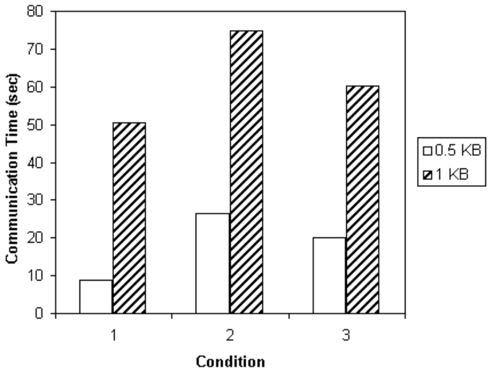
In the second experiment, a bursty but conformant (using TCP) distributed FFT computation has to compete with a non-conformant UDP stream. The FFT uses nodes m2, m3 and m4, and the UDP stream uses nodes m5 and m4, as before. In Figure 10(b), we show how the FFT communication time is affected by the UDP flow under different conditions. We present results for two FFT data sizes, 0.5K and 1K. As a base case, in Experiment 1, the UDP flow is idle. In Experiment 2, the UDP flow is active and as a result, the FFT communication times are dramatically increased. Finally, in Experiment 3 we deploy the delegates, which correctly identify the UDP flow and drop its packets. This reduces the communication times for the FFT compared to Experiment 2.

4.5 Virtual Private Network Service

A Virtual Private Network(VPN) [11] is an overlay network that is created within an existing physical network. It is virtual in that there is no separate physical infrastructure for this network and multiple virtual networks can run simultaneously within one network. It is private in that the data traffic belongs to one virtual network



(a) TCP versus UDP



(b) FFT versus UDP

Figure 10: Conformant TCP and FFT competing with non-conformant UDP

must not be observed by other virtual networks that are sharing the same infrastructure.

In a related project [?], we used Darwin as the basis for the development of a VPN service that allows per-VPN customization of control protocols and QoS support. A basic prototype has been running since October 1999. The per-VPN customizability relies on both the delegates and hierarchical resource management. Specifically, hierarchical resource management makes it possible to isolate the resources and traffic of different VPNs and to realize per-VPN QoS by having each VPN define its own subtree in the resource hierarchy. Delegates, on the other hand, allow control plane protocols specific to a VPN to be deployed dynamically.

To invoke the VPN service, users specify the topology of their VPN (e.g., virtual links and routers, bandwidth information, client networks that should be served by the VPN) and they specify the control plane protocols. The VPN service uses this information to determine what resources it needs in the network, and it then formulates a request for Beagle to set up VPN links and the control plane protocols (as delegates). One of the more difficult problems raised by this application is how to best deal with routing, given that different VPNs might use different routing protocols. We simply use a different table for each VPN in our current implementation [?]. Each VPN routing table is maintained by a routing daemon that implements RIP. The routing daemon currently coded in C is viewed as a special routing delegate. Once the VPN has been established, the user is notified that it can start sending data. Data packets originated from end-users belonging to a VPN are identified at the edge routers using our classifier. The edge router inserts a VPN ID into the packet header, and inside the network, core routers use flow classification based on the VPN ID to identify which VPN the packet belongs to. The data forwarding plane can then handle the packet appropriately, e.g., in terms of routing and QoS. IPsec is used for privacy.

5 Delegate Security

The programmable nature of an active network brings legitimate safety and security concerns.

The safety issues brought to the routers by delegates include general code safety concerns, various types of denial of services attacks, and privacy and security concerns. For example, a malicious or badly-implemented delegate can pose various threats to the router ranging from router state corruption to machine crash. This problem is being addressed by many research groups, and solutions from a variety of runtime mechanisms (e.g. Java sandboxing) to compile time mechanisms (e.g. proof carrying code [18]). In this section, we focus on security issues related to traffic management and control. Examples of threats include unauthorized use of bandwidth allocated to other flows, and redirecting or dropping the traffic belonging to other users. We explain these problems by examining three classes of increasingly more powerful delegates.

The first class of delegates can only control local resource allocation, i.e., it can modify the classifier and scheduler on the output ports. The primary concerns are: 1) a delegate manipulates traffic flows that it is not responsible for; 2) a delegate changes the scheduler parameters for resources belonging to other users. Our solution is based on associating delegates with specific flows and nodes in the resource tree, as shown in Figure 3. When Beagle sets up the flows and the delegates, it provides node operating system with information about what flows and resources the delegate can control. The node OS can store this information in the form of access control list. At runtime the node OS can use the information in the call (e.g., the flow_id) to make sure a delegate call is valid by consulting the access control list. While a simple "all or nothing" access control mechanism is sufficient for the examples we have considered so far, it is probably useful to have finer grain access control over the range of actions that a delegate can take. Resource management operations can be subdivided into more levels, e.g., monitoring traffic only, modifying the QoS parameters, changing the structure of a subtree (adding, deleting nodes). Note that this is similar to how the Unix file system does access control.

The second class of delegates can affect flow forwarding by changing the states in the forwarding engine. We still have the above mentioned concerns, i.e., delegates manipulate flows that they are not responsible for and change flow redirecting methods associated with other flows. The solution using access control mechanisms can be used to prevent a delegate from manipulating other flows. However, a new set of problems comes into picture even when a delegate only operates on the flows it is in charge of: it can use its own flows to pose potential threats to other routers or end-hosts in the network. For example a delegate can launch a denial-of-service attack by redirecting its own flows using tunneling to a victim server; a delegate can reroute its flow to critical links to cause congestion; or IP spoofing can occur if a delegate is allowed to add arbitrary tunnel headers to its flows.

The key insight to address these new problems is to constrain the delegate actions within the "virtual network" that corresponds to the service being deployed. The "virtual network" defines the links and routers that will be used by this service. Beagle reserves link resources on the output ports of these routers and installs delegates. Beagle owns the global view, i.e., the list of output inferfaces on the routers, the client networks, etc., of the "virtual network" for each service. To make sure that the delegates actions are within the "virtual network", at setup time Beagle passes its global view to the node OS on individual routers. With this information, node OS at runtime will then be able to check that whether or not the actions to be taken by delegates conform to the global view.

So far we have assumed that all delegates are installed by Beagle, and that Beagle can be trusted to provide the appropriate access control information. In a richer delegate model, delegates can create other delegates under certain conditions. This would for example make it possible to deploy a signalling protocol that can create delegates as a delegate. This would allow the VPN service to be used to create hierarchies of VPNs. A similar but simpler example is that a delegate is allowed to "delegate" authority to another delegate. Supporting this model will require a richer specification of the delegate authorities than in the earlier models.

Our effort in this area has focused on the first model, which is sufficient for many examples. We will refine and implement our solutions for the other models in the near future.

6 Related Work

In an active or programmable network, the functionality of the network can be extended on the fly, either through the use of active packets that carry the code that should be used to handle the packet, or by dynamically installing extensions on the routers [23]. Control delegates are an example of active networking applied to the control plane. The Darwin delegate facility is based on active extensions for performance and efficiency reasons: we do not want to pay the penalty of invoking custom code for every packet or even a very large number of packets. Instead, delegates execute out-of-band relative to the data flow and are executed asynchronously in response to specific events. As the examples in this paper show, many routing functions are well-suited to this style of invocation. The drawback of this approach is that there is a higher cost associated with installing active code, compared with active packets. This suggests that Darwin delegates may be less efficient than, for example, code groups in ANTS, for short-lived, "datagram-like" applications. A number of good overviews of active and programmable networking research are available in the literature [7, 22]. In this section we focus on the research efforts that are most closely related to Darwin delegates.

DARPA sponsored Active Net Node OS working group defined an "active node" architecture [6]. The Active Node (AN) architecture supports an Execution Environment (EE) that can execute Active Applications (AA), which can be either active packets or extensions. The delegate runtime environment can be viewed as a EE executing in the control plane of the router. It handles "control" packets, while regular "data" packets follow the "cut through" channel, i.e., they bypass the EE. While Darwin delegates follow the AN architecture, there is one difference. We have focused on the router programming interface, RCI, which is an interface that is not explicitly present in the more general AN architecture.

The ARP(Active Reservation Protocol) project [3] at ISI is developing a framework for implementing and deploying complex network control functions using an active network approach. Within this framework, new and modified services can be dynamically installed. Similar to Darwin, control protocols in ARP use an API called the protocol programming interface (PPI) to control the data forwarding path. So far ARP has focused on managing versions of traditional control protocols for routing and signaling, and not so much on support for control protocols that are customized for specific users. Also, protocols in ARP are network-wide, i.e., their actions are not restricted to specific sets of flows.

The Pronto [14] project provides a platform to support network programmability. Pronto supports a number of active networking models, including the one emphasized in Darwin, i.e. control plane programmability. One differences between the projects is that some Pronto calls imply a stronger coupling between data and control plane. For example, Pronto achieves frame dropping by having the control plane identify the packets that should be dropped, while Darwin relies on a classifier in the data plane to identify those packets. The Darwin interface is also slightly richer, although there appears to be no reason why the additional calls could not be incorporated into Pronto.

There has recently been a lot of work as part of the Xbind [16, 25] and TINA [10, 24] efforts to define a service-oriented architecture for telecommunication networks. While these efforts have similar goals to Darwin, there are also differences. First, services envisioned by Xbind and TINA are mostly telecommunications-oriented, while Darwin and delegates target a broader set of services. Second, while the focus of both TINA and Xbind is on developing an open service framework, the focus of Darwin is on developing *mechanisms*, e.g. for resource management, that can be *customized* to meet service-specific needs.

In part as a result of efforts such as XBind, a standardization effort was started to define an industry standard for an interface for programmable networks. This work is done in the context of the IEEE P1520 working group[1]. Clearly this effort is similar to the Darwin RCI, and we hope that some of our results can feed into this standardization effort. Darwin is also looking at the broader question of how to structure and build a system that uses this interface effectively. Another related standards effort of a control interface for router is GSMP[19], but it is much more narrow in scope.

7 Conclusion

In this paper we presented a programmable network architecture, in which the control plane functionality of the router can be extended using "delegates", code segments that implement customized traffic control policies or protocols. Delegates affect how routers treat the packets belonging to a specific user through a well-defined programming interface, Router Control Interface, or RCI. This open architecture offers opportunities to a broader community including third-party software vendors and value-added service providers, to develop applications for routers.

The node architecture was implemented in the CMU Darwin system, and we described a number of delegate examples that were executed on a testbed to demonstrate a range of applications can receive benefits via such a system. The applications include congestion control for video streaming and load balancing for client-server applications. Examples on tracking down non-conformant traffic sources and dynamically deploying VPN services gave a flavor of specific network control policies can be created using such a programmable infrastructure. While none of the examples provides necessarily the best, or even a complete, solution to these problems, they do illustrate that a rich set of traffic control and management services can be conveniently deployed through the system we provided.

We will extend our work in the directions of generalization of the architecture, performance evaluation in wide area networks and dealing with delegate security issues.

References

[1] J. Biswas, J. Huard, A. Lazar, K. Lim, S. Mahjoub, L. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein. The ieee p1520 standards initiative for programmable network interfaces. *IEEE Communications Magazine*, 36(10):64–70, October 1998.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. AN Architecture for Differentiated Service, December 1998. IETF RFC 2475.

[3] Bob Braden. Active Reservation Protocol (ARP), December 1998. Abstract at URL http://www.isi.edu/div7/ARP/.

[4] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview, June 1994. Internet RFC 1633.

[5] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource Reservation Protocol (RSVP) – Version 1 Functional Specification, September 1997. IETF RFC 2205.

[6] Architectural framework for active networks, August 1998. Available over the net through URL http://www.cc.gatech.edu/projects/canes/arch/arch-0-9.ps.

[7] Andrew Campbell, Herman De Meer, Michael Kounavis, Kazulo Miki, John Vincente, and Daniel Villela. A survey of programmable networks. *IEEE Communications Magazine???*

[8] Prashant Chandra, Allan Fisher, Corey Kosak, and Peter Steenkiste. Network Support for Application-Oriented Quality of Service. In *Proceedings Sixth IEEE/IFIP International Workshop on Quality of Service*, pages 187–195, Napa, May 1998. IEEE.

[9] Prashant Chandra, Allan Fisher, and Peter Steenkiste. Beagle: A resource allocation protocol for an application-aware internet. Technical Report CMU-CS-98-150, Carnegie Mellon University, August 1998.

[10] F. Dupuy, C. Nilsson, and Y. Inoue. The tina consortium: Toward networking telecommunications information services. *IEEE Communications Magazine*, 33(11):78–83, November 1995.

[11] P. Ferguson and G. Huston. What is a VPN? In *OPENSIG'98 Workshop on*

Open Signalling for ATM, Internet and Mobile Networks Workshop, Toronto, October 1998.

[12] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

[13] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[14] The pronto platform - a flexible toolkit for programming networks using a commodity operating system, October 1998. Unpublished draft.

[15] javasoft.sun.com. The Source for Java(TM) Techonology. http://www.javasoft.com/.

[16] A. Lazar, Koon-Seng Lim, and F. Marconcini. Realizing a foundation for programmability of atm networks with the binding architecture. *IEEE Journal on Selected Areas in Communication*, 14(7):1214–1227, September 1996.

[17] Dong Lin and Robert Morris. Dynamics of Random Early Detection. In *Proceedings of the SIGCOMM '97 Symposium on Communications Architectures and Protocols*, pages 127–137, Cannes, August 1997. ACM.

[18] George Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 229–243. Usenix, October 1996.

[19] P. Newman, W. Edwards, R. Hinden, E. Hoffman, F. Ching Liaw, T. Lyon, and G. Minshall. Ipsilon's general switch management protocol specification version 2.0, March 1998. RFC 2297.

[20] K. Nichols, V. Jacobson, and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet, July 1999. IETF RFC 2638.

[21] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service. In *Proceedings of the SIGCOMM '97 Symposium on Communications Architectures and Protocols*, pages 249–262, Cannes, September 1997. ACM.

[22] David Tennenhouse, Jonathan Smith, David Sincoskie, David Wetherall, and Gary Minden. A survey of active networking research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[23] David Tennenhouse and David Wetherall. Towards and active network architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[24] Tina consortium. http://www.tinac.com.

[25] Project X-Bind. http://comet.ctr.columbia.edu/xbind.